

SL Converter Manual

2013/06/24

Version 2.0

**The content of this document is highly confidential
and should be handled accordingly.**

Confidential

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

Table of Contents

1	Introduction	4
1.1	Purpose of SL Converter	4
1.2	Target Users of SL Converter	4
1.3	Benefits of Using SL Converter	4
2	How to Use the Tool	5
2.1	How to Run the Tool	5
2.2	Command Options	5
2.3	Using as a Library	8
3	Precautions When Converting to Shaders for Wii U	9
3.1	Differences in Matrix Definitions	9
3.2	Support for DirectX Effect Files	9
3.3	Memory Capacity for Constants	9
3.4	Passing Parameters Between Shaders	9
3.5	Uniform Variable Binding	10
3.6	Attribute Bindings and Configuration Files	11
3.7	Batch Processing	12
3.8	Batch Processing of Effect Files	12
3.9	Location Binding in UniformBlock	13
3.10	Binding and Configuration Files for UniformBlock	14
3.11	Specifying the Sampler Type with a Config File	14
3.12	Using fetch4	15
3.13	Support for Multithreading	16
4	Known Issues	17
4.1	sampler Variable Handling	17
4.2	Buffer Variables	17
	Revision History	18

Code

Code 3-1	Example of Using Semantics	9
Code 3-2	Allocating Uniform Variables in HLSL	10
Code 3-3	Example Configuration File	11
Code 3-4	Sample Effect File	12
Code 3-5	Sample Effect File	13
Code 4-1	sampler Example	17

1 Introduction

1.1 Purpose of SL Converter

SL Converter is a tool to convert DirectX HLSL (proprietary high-level shader language) files to GLSL (OpenGL, a high-level shader language based on C programming syntax) for use with the Wii U. This allows you to use shaders developed in HLSL with the Wii U.

1.2 Target Users of SL Converter

The following developers are considered target users of SL Converter.

- Developers who want to port existing shader files written in HLSL to the Wii U.

1.3 Benefits of Using SL Converter

SL Converter offers the following advantages.

- Support for batch conversion of a large number of shader files.
- Support for DirectX effect file format (.fx). Converts shaders for each internal technique.
- Support for multi-threaded, high-speed shader conversion.
- Support for Wii U proprietary GLSL extensions (GLSL Cafe Extension).
- Support to specify `VertexAttribute` positions in configuration file.
- Support for conversion handles matrices and other differences between HLSL and GLSL.

2 How to Use the Tool

2.1 How to Run the Tool

Execute the tool as follows.

- `SLConverter.exe [options] input-filename`

Specify the HLSL source shader file for *input-filename*.

You can also use wildcards, such as `*.fx` or `*.hlsl`. Options are case-sensitive.

Note: The Cafe compiler uses the shader compiler in the directory specified by the `CAFE_ROOT` environment variable.

If `CAFE_ROOT` is misconfigured, the compile will not run even when using the `-v` option to specify Cafe.

2.2 Command Options

The tool has the following options. You can specify multiple options at once.

- `-b`

Disables GLSL Cafe Extension uniform binding. (Enabled by default.)

- `-c <configuration file name>`

Specifies the configuration file name. (The default is to use no configuration file.)

- `-D <macro name>`

Defines a preprocessor macro. You can optionally set a value.

Example with no values: `-DMYDEFINE`

Example with value appended: `-DMYDEFINE=10`

If you want to define multiple macros, you need to apply the `-D` option for each one.

Example: `-DMYDEFINE1 -DMYDEFINE2=3`

- `-e <main function name>`

Specifies the shader's main function.

- `-s <vs, ps, gs>`

Specifies the shader type. Specify one of the following.

- `vs` - Vertex shader (default)
- `ps` - Pixel shader
- `gs` - Geometry shader

- `-i`
Uses the HLSL source shader's semantic name for the varying variable name passed from the vertex shader to the pixel shader.
- `-I <include directory name>`
Specifies the directory that contains files to include when compiling the shader. Separate multiple directory entries using a ";" (semicolon).
- `-h`
Displays a help message.
- `-l`
Disables location specification for varying variables. (Enabled by default.)
- `-m`
Disables changing the order of matrix `mul` operations. (Enabled by default.)
- `-o <name of GLSL file to output>`
Specifies the output file name. (By default, this option outputs the same file name as the input file name, only changes the extension to `.glsl`.)
You can also specify a pattern for the output file.
 - `$0`: Input HLSL file name without extension.
 - `$1`: Shader method name.
 - `$2`: Directory of input HLSL file.
 For example, specify `$0_$1.glsl` to convert the shader called `MainVS` in the `sample.fx` file into an output file named `sample_MainVS.glsl`.
- `-p`
Uses the texture coordinates in the pixel shader (semantic name of `TEXCOORD`) as `gl_PointCoord`.
- `-v <Cafe, GL, None>`
Runs a test compile of the converted shader. Specify one of the following.
 - `Cafe` - Use the compiler in Cafe SDK. (Default.)
 - `GL` - Use the standard GLSL compiler.
 - `None` - Run test compile.
- `-t`
Uses multi-threading to compile multiple shaders in parallel. (Disabled by default.)

- `-V <true, false>`

Enables or disables display of log messages. (Enabled by default.) This option is similar to the `-v` verbose option for Linux/UNIX ftp commands.)

- `-r`

Search files recursively. (Disabled by default.)

- `-n`

Converted shaders are not compiled when this option is specified.

- `-z`

Generates a code that uses the Vertex Shader to convert the z projection space so that it shifts from `[-1,1]` to `[0,1]`.

- `--builtin<VALUE>`

When the function name specified by this option is converted to GLSL, the name will be output unchanged, without unnecessary conversions.

Example: `--builtin "float4 texture4(sampler xxx, float2 pos);"`

In the example above, the function called `texture4` is defined within HLSL, but is ignored when converted to GLSL and output as a function already in GLSL.

- `@<list file>`

Specifies a file listing the files to convert. For example, enter the command `SLConverter.exe @listfile.txt` where the `listfile.txt` content is as shown below:

```
o MyShader1.hlsl#main#vs#MyShader.glsl
o MyEffect.fx
```

Note that the `@<list file>` command produces the same results as separately running the following two commands:

```
o SLConverter.exe MyShader1.hlsl -e main -s vs -o MyShader.glsl
o SLConverter.exe MyEffect.fx
```

Converting multiple files with just one process invocation speeds up processing.

- `--maptype NAME=TYPE`

Can force the conversion of a sampler specified by `NAME` to a specific sampler type.

For example: Using the option `-maptype ShadowSampler=sampler2DShadow` converts the sampler named `ShadowSampler` into uniform `sampler2DShadow ShadowSampler;` when converted by GLSL. (Ordinarily it becomes `sampler2D`.)

- `-savegsh`

Specify this option to output the compiled resulting binaries to the `gsh` file.

2.3 Using as a Library

Starting with version 1.3, you can use `SLConverter` as a library. (You can call this from the .NET programming languages.)

Follow the steps below to use as a library.

1. Reference the `SLConverter.exe` file from the .NET project.
2. Generate an `SLConverter` object.
3. Decide which compiler to use.
4. Turn the converter flags on or off as appropriate for your preferences.
5. Convert using the converter's `ConvertFromText` or `ConvertFromFile` functions.

Detailed specifications are available in the SL Converter API documentation.

See the source code of the `SLConverterSamples.sln` demo for a detailed sample.

3 Precautions When Converting to Shaders for Wii U

3.1 Differences in Matrix Definitions

In HLSL, matrix definitions such as `float4x4` are `row × column`, but in GLSL, they are `column × row`.

Simply swapping the row and column causes the binding index to be off when using the Wii U explicit uniform binding feature.

SL Converter switches the calculation order for matrices and vectors for conversion, so the application does not need to do any matrix configuration.

Use the `-m` option to disable order switching, as it is enabled by default.

3.2 Support for DirectX Effect Files

DirectX has files called "effect files" that allow you to change the shader or compiler used for each technique or rendering path. OpenGL generally does not have such a feature.

SL Converter supports DirectX effect files. When the input file is an `.fx` file, SL Converter converts the shaders used for each technique and path described in the effect file into separate output files.

3.3 Memory Capacity for Constants

In HLSL, there are 256 uniforms for `float4`, as well as 16 tables each for `bool` and `int` constant uniforms. GLSL Cafe Extension only has 256 uniforms for `float4` (when not using `UniformBlock`).

A shader cannot be reproduced in GLSL Cafe Extension if there are more than 256 uniforms total in the HLSL source for `float4`, `bool`, and `int` types.

SL Converter does not support such cases. The developer must revise the HLSL source code.

3.4 Passing Parameters Between Shaders

In HLSL, parameters are passed between shaders by means of semantics. Because GLSL has no semantic feature, name resolution is used instead.

Code 3-1 Example of Using Semantics

```
struct VSOut
{
    float2 texCoord : TEXCOORD0;
};

VSOut VS_Main(...)
{
    VSOut Out;
```

```

        :
        :
        return Out;
    }

    struct PSIn
    {
        float2 uv : TEXCOORD0;
    };

    float4 PS_Main( PSIn In ) : COLOR
    {
        :
        :
    }

```

In the HLSL example above, the `VSOut` member `texCoord` and the `PSIn` member `uv` have different names, but parameter passing resolves as they are the same semantic.

GLSL has no concept of semantics, therefore you must have the same varying (or `in`, `out`) variable names.

Use the `-i` command option with SL Converter to assign variable locations based on the semantic in the original HLSL source.

Consequently, even though the names might be different in the vertex shader and pixel shader, the GLSL will be output with parameter passing resolved, provided the semantics are the same.

3.5 Uniform Variable Binding

HLSL can explicitly assign uniform variables.

Code 3-2 Allocating Uniform Variables in HLSL

```
float4x4 u_modelMtx : WORLD : register(c0)
```

Plain GLSL does not have such a feature, but GLSL Cafe Extension has an extension that allows you to explicitly specify a uniform's location. SL Converter's default behavior is to convert shaders while allowing this extended feature.

In HLSL, you can assign explicit locations for some, all, or none of the uniform variables. However, in GLSL for Wii U you must assign a location for EVERY uniform variable if you choose to manually assign explicit locations at all.

When converting HLSL to GLSL and the HLSL source includes a variable for which no register number is explicitly specified, the converter automatically assigns an unused register number.

Use the `-b` option to disable GLSL Cafe Extension uniform binding, as it is enabled by default.

3.6 Attribute Bindings and Configuration Files

In HLSL, vertex shader input is bound to vertex data set by the application using semantics, but GLSL has no such system.

Therefore, SL Converter uses configuration files that allow you to explicitly specify which HLSL source vertex shader input to assign to which designated GLSL attribute.

Configuration files must be in XML format, as shown in the example below.

Code 3-3 Example Configuration File

```
<?xml version="1.0" encoding="utf-8"?>
<config>
  <layout semantic="POSITION" name="a_Position" location="0" />
  <layout semantic="POSITION0" name="a_Position0" location="0" />
  <layout semantic="TEXCOORD" name="a_TexCoord" location="5" />
  <layout semantic="TEXCOORD0" name="a_TexCoord0" location="5" />
  <layout semantic="TEXCOORD1" name="a_TexCoord1" location="6" />
</config>
```

`<config>`: This is the root element, and can contain single or multiple `<layout>` elements.

`<layout>`: Use this element to specify the following attributes.

- `semantic`

The vertex shader input semantic in the HLSL source.

- `name`

The designated attribute name in the GLSL.

- `location`

The designated attribute location in the GLSL.

For instance, `<layout semantic="POSITION" name="a_Position" location="0" />` indicates that the vertex shader input with the `POSITION` semantic in the HLSL source is a designated attribute bound with the name `a_Position` and a location of 0 in the GLSL.

You can also specify a position using a string instead of a number. Use code such as `<layout semantic="POSITION" name="a_Position" location="LOC_POS" />` to output the following in the converted shader.

```
layout(location = LOCATION_POSITION) in vec4 a_Position;
```

This assumes something like `#define LOCATION_POSITION 0` on the user side, just after `#version` or `#extension`.

This allows you to ensure consistency between your user-side C++ code and the shader-side locations.

3.7 Batch Processing

Depending on the project, you might have to convert several hundred, or even thousand, HLSL shaders into GLSL.

Typing the commands to manually convert each file takes a long time, so instead you can use wildcards in file names you input into SL Converter to process multiple files in a batch.

For instance, you can use the command line `C:>SLConverter *.hlsl` to convert all files with the `.hlsl` file name extension.

Alternately, you can use the `-t` option to significantly shorten the time needed to process large files by using multi-threading to compile multiple files in parallel.

Example: `C:>SLConverter -t *.hlsl`

Use the `-r` option to search through directories recursively. Alternatively, you can do so by specifying `**` as the filename.

Example: `C:>SLConverter -r *.hlsl`

Example: `C:>SLConverter **.hlsl`

Each of these examples searches recursively for files with the `.hlsl` extension, and then converts all of them.

3.8 Batch Processing of Effect Files

When the main entry function is not specified or when the input file is a DirectX effect file, SL Converter splits shaders used for each technique and path described in the file into a separate file and converts that.

Code 3-4 Sample Effect File

```
// sample.fx

Technique Tech0
{
    Pass Pass0
    {
        VertexShader = compile vs_1_1 Tech0_VS();
        PixelShader  = compile ps_1_1 Tech0_PS();
    }
}

Technique Tech1
{
    Pass Pass1
    {
        VertexShader = compile vs_1_1 Tech1_VS();
        PixelShader  = compile ps_1_1 Tech1_PS();
    }
}
```

```
}
```

Inputting the above effect file outputs the following GLSL files:

- Sample_Tech0_VS.glsl
- Sample_Tech0_PS.glsl
- Sample_Tech1_VS.glsl
- Sample_Tech1_PS.glsl

You can also output any shader from an input file into any output file. In addition, you can specify multiple shaders and multiple output files.

- <Filename>#<entrypoint>#<stage>#<output file>

Use the “#” (hash) mark as the delimiter between fields to convert multiple shaders as a batch.

Code 3-5 Sample Effect File

```
SLConverter.exe sample.fx#SimpleVS#vs#vs.glsl sample.fx#SimplePS#ps#ps.glsl
```

The sample command above produces the same results as separately running the following two commands:

- SLConverter.exe sample.fx -e SimpleVS -s vs -o vs.glsl
- SLConverter.exe sample.fx -e SimplePS -s ps -o ps.glsl

Batch conversion using a single command speeds up processing by eliminating the overhead of launching multiple commands.

3.9 Location Binding in UniformBlock

In HLSL, you can explicitly apply an offset value to each variable in a constant buffer as follows.

```
cbuffer cbChangesEveryFrame
{
    matrix World : packoffset(c0);
    float hoge : packoffset(c4.x);
    float4 vMeshColor : packoffset(c5);
};
```

The converted GLSL will be as follows.

```
uniform cbChangesEveryFrame {
    layout(location = 0) mat4 World ;
    layout(location = 64) float hoge ;
    layout(location = 80) vec4 vMeshColor ;
};
```

However, even though it is possible to specify only the intermediate portion of a register in HLSL as in the following code, it is not possible to do so in GLSL, so SL Converter does not support this kind of specification. In this case it is necessary to rewrite the original HLSL shader

```
cbuffer cbChangesEveryFrame
{
    matrix World : packoffset(c0);
    float hoge : packoffset(c4.y); ←
    float4 vMeshColor : packoffset(c5);
};
```

3.10 Binding and Configuration Files for UniformBlock

With SL Converter, you can explicitly specify the assigned location for a GLSL file after the conversion of an HLSL file's constant buffer location.

For example, consider the following settings written in the configuration file,

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    <cbuffer register="colors" binding="COLORS" />
</config>
```

```
// HLSL prior to conversion
cbuffer cbColors : register( colors )
{
    float4 color0;
    float4 color1;
};
```

This will result in the following GLSL file after conversion.

```
layout(binding = COLORS) uniform cbColors {
    vec4 color0;
    vec4 color1;
};
```

It is assumed that `COLORS` is set with a `#define` statement, etc., in the user application.

3.11 Specifying the Sampler Type with a Config File

By including instructions like those that follow in a config file, you can force the conversion of a specific sampler into the desired type.

```
<map name="shadowSampler" type="sampler2DShadow"/>
```

In the example above, if the sampler named `shadowSampler` is in HLSL, would be converted by GLSL into:

```
uniform sampler2DShadow shadowSampler;
```

The same thing can be done with the `-maptype` command option.

3.12 Using fetch4

A function called `texture4` can be used with the shader in an actual Wii U system. This function is able to load the four-pixel information adjacent to a one-time texture fetch. (It is mainly used as the shader for components like shadow maps.)

Even in the case of DirectX, there is hardware that allows the same types of functions to be used. However, this can be used by entering special values to the sampler API, not by the use of HLSL. Hence, because you will not know which texture function should be handled as `texture4` by merely looking at the HLSL shader, special options have been provided in SL Converter.

First, the following code must be written for HLSL, the source of the conversion.

```
sampler Texture4Sampler;

// Define texture4 for HLSL
// This function will be ignored by SL Converter and will directly
// use texture4
float4 texture4(sampler mySampler, float2 position)
{
    return tex2D(mySampler, position);
}

float4 PS() : COLOR
{
    float4 texVal = texture4(Texture4Sampler, uv);
    :
    :
}
```

Next, the following settings must be added to the config file that is used.

```
<builtin>float4 texture4(sampler xxx, float2 pos);</builtin>
```

There is also the method of adding `--builtin "float4 texture4(sampler xxx, float2 pos);"` to the command line option.

In this manner, when SL Converter does its conversion, the following section is ignored.

```
float4 texture4(sampler mySampler, float2 position)
{
    return tex2D(mySampler, position);
}
```

The `texture4` function is used as is as a function already defined by GLSL.

3.13 Support for Multithreading

When you are using SL Converter as a library and you want to use multithreading for distributed processing, Nintendo recommends creating multiple instances of the `SLConverter.Converter` class for each thread and using those.

However, if the properties for `SLConverter.Converter` are the same for all threads when run, the same instance of `SLConverter.Converter` can be used with multiple threads.

4 Known Issues

4.1 `sampler` Variable Handling

In HLSL, you can handle generically defined `sampler` variables as `sampler2D`, `samplerCUBE`, or `sampler3D` types. However, GLSL has no analogous mechanism. (You must explicitly specify the type, such as `sampler2D`.)

SL Converter converts variables defined as `sampler` in HLSL into `sampler2D` in GLSL.

Consequently, using a variable defined as `sampler` in the HLSL for some use other than `sampler2D` results in an incorrect conversion.

Code 4-1 `sampler` Example

```
sampler u_MyTex; // At this point, cannot tell if this is 2D, Cube, or 3D.
:
:
:
float4 SampleCube( samplerCUBE tex, float3 pos )
{
    return texCUBE(tex,pos);
}

:
:
:
float4 main(): COLOR
{
    float4 color = SampleCube(u_MyTex, float3(x,y,z));
    return color;
}
```

This limitation might be revised in a future release, but as of version 1.7 the `sampler u_MyTex;` portion of the above HLSL code must be explicitly replaced with `samplerCUBE u_MyTex;` for this to convert correctly.

4.2 Buffer Variables

In the DirectX 10 HLSL, you can define buffer objects such as `Buffer<float4>`, but this feature is not supported in SL Converter. Accordingly, attempting to convert HLSL that uses buffer objects will result in an error.

Revision History

Version	Revision Date	Category	Description
2.0	2013/06/24	Changed	- Added information to 2.2 Command Options. - Deleted Shader Model 4.0 from Known Issues. - Added 4.2 Buffer Variables to Known Issues.
1.9	2012/10/04	Added	Added 3.13 Support for Multithreading.
1.8	2012/08/09	Changed	- Added to 2.2 Command Options. - Added to 3.12 Using fetch4.
1.7	2012/07/05	Added	Added to 2.2 Command Options.
1.6	2012/06/20	Added	- Added to 2.2 Command Options. - Added 3.11 Specifying the Sampler Type with a Config File
1.5	2012/05/24	Added	- Added to 2.2 Command Options. - Added 3.10 Binding and Configuration Files for UniformBlock.
1.4	2012/04/25	Added	Added 3.9 Location Binding in UniformBlock.
1.3	2012/04/13	Added	2.3 Using as a Library
		Changed	2.1 How to Run the Tool Added <code>CAFE_ROOT</code> environment variable. 3.6 Attribute Bindings and Configuration Files Added description of string-based locations. 5 Changelog by Resource Moved to the <code>changelog.txt</code> file.
1.2	2012/03/29	Deleted	5 Changelog by Resource
1.1	2012/02/20	Changed	2.2 Command Options Added text. 3.1 Differences in Matrix Definitions Revised text. 3.7 Batch Processing Added text. 3.8 Batch Processing of Effect Files Revised text. 3.1 Differences in Matrix Definitions Revised text. 4 Known Issues Added chapter and text.
1.0	2012/01/27	—	Initial version.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

© 2012–2013 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.